TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MIKA KAARETKOSKI

ISOLATING SOFTWARE DEVELOPMENT ENVIRONMENTS

Master of Science thesis

Examiner: Prof. Kari Systä
Examiner and topic approved by the
Faculty Council of the Faculty of
Information Technology
on 30th July 2016

ABSTRACT

MIKA KAARETKOSKI: Isolating software development environments
Tampere University of Technology
Master of Science thesis, 50 pages
September 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Kari Systä
Keywords: isolation, development environments, virtualization, Nix

Software development processes are evolving constantly, aiming to more agile and efficient approaches. Agility demands ability to adapt to quick changes in requirements, which can lead to modifications in development environments. Managing environments for developing and testing has become an essential part of efficient development work. Developers should be able to modify environments safely without having to worry about disrupting something else in the system. Working on multiple projects at the same time without conflicting dependencies should also be possible. The amount of software dependencies in a modern application using open-source components is typically very large, so conflicts between projects are rarely avoided without any isolation between environments.

Virtualization is at a key role providing not only the needed isolation, but also the tools for managing the environments. Virtual machines and containers utilizing modern tools and large ecosystems for open-source software make creating and managing isolated development environments efficient. There are also other means besides virtualization to avoid conflicting environments, one of which is using a functional package manager that allows installing software components in isolation from each other.

In this thesis, different ways of creating an isolated development environment are discussed. Three approaches achieving different levels of isolation are presented; full isolation using virtual machines, namespace isolation using containers and component isolation using a functional package manager. The focus is on introducing the technologies and tools for isolating a development environment using a simple practical example for each approach. The approaches are compared on a high level on few relevant categories which are level of isolation, reproducibility, resource overhead, usability and support and availability. The thesis is concluded with a quick summary and some discussion about choosing between the isolation solutions.

TIIVISTELMÄ

MIKA KAARETKOSKI: Ohjelmistokehitysympäristöjen eristäminen
Tampereen teknillinen yliopisto
Diplomityö, 50 sivua
Syyskuu 2018
Tietotekniikan diplomi-insinöörin koulutusohjelma
Paine: Ohjelmistotuotanto
Tarkastajat: Prof. Kari Systä
Avainsanat: eristäminen, ohjelmistokehitysympäristöt, virtualisointi, Nix

Ohjelmistokehitysprosessit muuttuvat jatkuvasti kohti ketterämpiä ja tehokkaampia ratkaisuja. Ketteryys vaatii kykyä mukautua nopeasti muuttuviin vaatimuksiin, joka voi johtaa muutostarpeisiin kehitysympäristössä. Kehitys- ja testausympäristöjen hallinnasta on tullut olennainen osa kehitystyötä. Ympäristöjä pitää pystyä muokkaamaan turvallisesti, huolehtimatta muutoksen vaikutuksista muuhun järjestelmään. Yhtäaikainen työskentely useamman projektin kanssa tulee myös olla mahdollista, ilman ongelmia projektien komponenttiriippuvuuksien välillä. Komponenttiriippuvuuksien määrä nykyaikaisissa vapaan lähdekoodin komponentteja käyttävissä projekteissa on valtava, joten on vaikea välttyä riippuvuusongelmilta ilman kehitysympäristöjen eristämistä.

Virtualisointi on avainroolissa sekä eristyksen että ympäristönhallintatyökalujen tarjoamisessa. Nykytyökaluja ja laajoja avoimen lähdekoodin palveluja hyödyntävät virtuaalikoneet ja kontit tekevät eristettyjen kehitysmpäristöjen luonnista ja hallinnasta tehokasta. Virtualisoinnin lisäksi on myös muita tapoja välttää konflikteja ympäristöjen välillä, kuten komponenttien eristetyn asennuksen mahdollistavan, funktionaalisen paketinhallintatyökalun käyttö.

Tässä diplomityössä tutkitaan eri tapoja ohjelmistokehitysympäristön eristämiseen. Työssä esitellään kolme vaihtelevan eristystason tuottavaa menetelmää; täysi eristys virtuaalikoneen avulla, nimiavaruustason eristys kyttäen kontteja ja komponenttieristys käyttäen funktionaalista paketinhallintatyökalua. Pääpaino on eristämiseen käytettyjen teknologioiden ja työkalujen esittelyssä käytännön esimerkkejä hyödyntäen. Menetelmiä vertaillaan yleisellä tasolla muutamassa eri kategoriassa. Kategoriat ovat eristyksen taso, toistettavuus, resurssitehokkuus, käytettävyys, tuki ja saatavuus. Lopuksi aihe käydään lyhyesti läpi sekä keskustellaan sopivan eristysratkaisun valinnasta.

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CBSE	Component-based Software Engineering
CPU	Central Processing Unit
GB	Gigabyte
GUI	Graphical User Interface
HW	Hardware
IDE	Integrated Development Environment
I/O	Input/Output
MB	Megabyte
OS	Operating System
RAM	Random Access Memory
SSH	Secure Shell
SHA	Secure Hash Algorithm
UnionFS	Union File System
URL	Uniform Resource Locator
VM	Virtual Machine
VMM	Virtual Machine Manager
VDI	VirtualBox Disk Image
VMDK	Virtual Machine Disk
YAML	Yet Another Markup Language

1. INTRODUCTION

The software development process is changing continuously. From code and fix through waterfall to agile, CBSE and DevOps [40, 11], the development process is evolving not only because of experience and increased understanding of efficient software development work, but also because of technological advancements.

Virtualization is a technique that has improved the IT infrastructure and software development greatly over the last ten years. It has enabled cloud computing, the main building block of the modern As-a-Service business models. Introduction of the cloud has made the internet more available, reliable and useful. The vast amount of open-source projects available allow code reusability on a larger scale than ever before. As more and more open-source code is available the more efficient producing new applications becomes.

However, being an efficient programmer requires more than being able to mix and match code well. Knowing and using the modern tools to help maximize one's productivity is vital. One key aspect in development work is managing environments for development and testing. When code is heavily reused, the amount of dependencies in environments gets larger and the amount of possible conflicts increases. Working on multiple projects with different requirements without any isolation of development environments becomes very cumbersome. Virtualization has enabled many powerful tools that among other benefits help to avoid the interference between environments. Tools like VirtualBox, Vagrant and Docker allow creation and management of isolated development and testing environments, which in addition to solving the problem with conflicting environments, offer other benefits. Reproducibility and efficient sharing of development setups are examples of advantages enabled by isolation. Isolation also adds flexibility into development work by allowing easier testing with different versions and configurations.

Non-conflicting environments are also possible without virtualization, using a specific package manager. Nix is a functional package manager with features that enable installing software components in isolation. The component isolation provided by Nix is based on its functional model for building software and using unique hashes in

component-names. As a non-virtualized, native solution, Nix has a smaller footprint in terms of resources, but also a weaker level of isolation compared to virtualization techniques.

This thesis studies different tools and techniques for isolating software development environments. Three different approaches for isolation are introduced. The underlying technologies and practical examples with relevant tools are presented for each of the solutions. Virtualization enables two of the solutions; virtual machines and containers. The solution using virtual machines is leveraging VirtualBox hypervisor and a VM management tool called Vagrant, while container-approach is based on Docker and Docker Compose. The third solution is purely based on Nix and its built-in features. The solutions are compared on few relevant categories; level of isolation, reproducibility, resource overhead, usability and support and availability. The goal of this work is to provide an introduction to different isolation methods, discuss how the solutions compare and how they can be useful in development work.

The structure of the thesis is following. Chapter 2 defines isolation and reasons behind it. An overview on virtualization and related techniques is given in Chapter 3. As an alternative to virtualization, Nix is introduced in Chapter 4. Isolating development environments and different approaches to it are discussed on a general level in Chapter 5, after which practical examples with relevant tools are presented in Chapter 6. The approaches are compared on a high level in Chapter 7 and finally, the thesis is concluded with some discussion on the subject.

2. BACKGROUND

2.1 Defining isolation

Isolation in general means that the environment should be self-contained and have no external dependencies. No external state or process should affect the functionality of the isolated environment. In practice, this kind of absolute isolation and self-containability is impossible to achieve as the the isolated environment is always dependent on the host machine at some level. E.g. virtual machines are dependent on the hardware virtualization software, which is depending on the host OS. Likewise, containers depend on the container platform running on the host OS. It is not useful to consider these lower level dependencies in the context of this work, so they are ignored. The isolation discussed in this work is looked mainly from a point of view of software components, so component isolation is considered the minimum requirement for an isolated environment. Component isolation means that software components in one environment should not be affected by changes in another environment.

This thesis introduces three different solutions to isolating a development environment. Two of the solutions, virtual machines and containers, are enabled by virtualization. The third approach is enabled by a functional package manager called Nix. Each of the three approaches produce a different level of isolation, all of which are fulfilling the minimum requirement of component isolation. Virtual machines provide the highest level of isolation, named full isolation, as they can be considered fully isolated from the host OS. Isolation by containers is achieved by a Linux feature called namespaces, hence it is called namespace isolation. Namespace isolation depends on the shared host kernel functionalities, so it is not as separated from the host as a virtual machine. The third isolation level is the minimum requirement, component isolation, and it is provided by Nix. The isolation levels can be thought as nested, meaning that a full isolation is inherently providing namespace- and component isolation and namespace isolation is inherently providing component isolation. There are different features at each level, and the highest level of isolation is not necessarily the best for development environments. These characteristics will be discussed more in Chapter 7.

2.2 Reasons to isolate

Todays development environments are expected to satisfy the needs of a modern, more agile software development process. Open source components are used on a very large scale [24] and with the reuse of these components comes a vast amount of software dependencies. There are lots of package and dependency managers available [41], but most of these do not provide mechanisms to avoid conflicts between environments. There are good solutions for some languages and ecosystems, like npm for JavaScript and Bundler for Ruby. However, with the vast amount of different tools and technologies used in modern projects, isolation for a stack consisting of a variety of languages is needed.

There are also other benefits related to isolation besides avoiding conflicts between environments. Isolation is in many ways useful for a developer. For one, isolation helps to create reproducible environments. Developers should be able to share, remove and reproduce environments easily and reliably. They also should have a possibility to easily switch between projects, being sure nothing previously configured or installed does not affect the current environment. These properties are reliably achievable only by using isolation. Another benefit is flexibility in testing with different versions and configurations. Isolated environment is disposable so one does not have to worry about the state of the environment, and can experiment freely.

The main goal in isolating development environments is to provide ability to work simultaneously on multiple projects with different development stacks without having to worry about that configurations done in one environment might have an affect on another environment. Other aforementioned benefits that come as a side-product are important factors when deciding between isolation options.

3. VIRTUALIZATION BASICS

A definition of the verb virtualize is "to transform something into an artificial computer-generated version of itself which functions as if it were real" [30]. In short, virtualizing is a way to create a usable abstraction of a device or resource. One of the most basic examples of this is a virtual memory in operating systems. Virtual memory is a memory management technique that extends RAM address space to hard drive, which effectively allows more memory to be allocated than physically available on RAM [3]. Therefore, virtual memory address space in hard drive is virtualized RAM.

Virtualization can be achieved on many levels of abstraction. Applications, hardware components, servers, networks, systems and services are all virtualizable using current techniques. Virtualization types can be categorized in multiple ways depending on the perspective, but in this thesis only the types that are closest to the subject of isolating development environments are discussed in more detail. Those are hardware virtualization and containerization also known as operating-system-level virtualization. Focus is on x86 CPU architecture which is most widely used in personal computers and servers today.

3.1 Hardware virtualization

One of the most common use-cases of virtualization is hardware virtualization. In hardware virtualization the computing resources of a physical machine are divided into virtual resources used by virtual machines to obtain e.g. better hardware utilization, isolation of environments and simultaneous use of resources. The physical machine is called a host, and the virtual machines are called guests or VMs. The VMs run on top of Virtual Machine Managers or VMMs, also known as hypervisors. The hypervisor performs scheduling and hardware resource allocation for all the virtual machines in the system. There are two types of hypervisors, Type-1 or native hypervisors and Type-2, hosted hypervisors. Native hypervisors are operating systems or kernels running directly on host machines hardware and use hardware to control the guest VMs, whereas hosted hypervisors are applications running on top

of an operating system and the guest VMs are run as a process on the host OS. The two hypervisor types are shown in Figure 3.1 [27, 32]

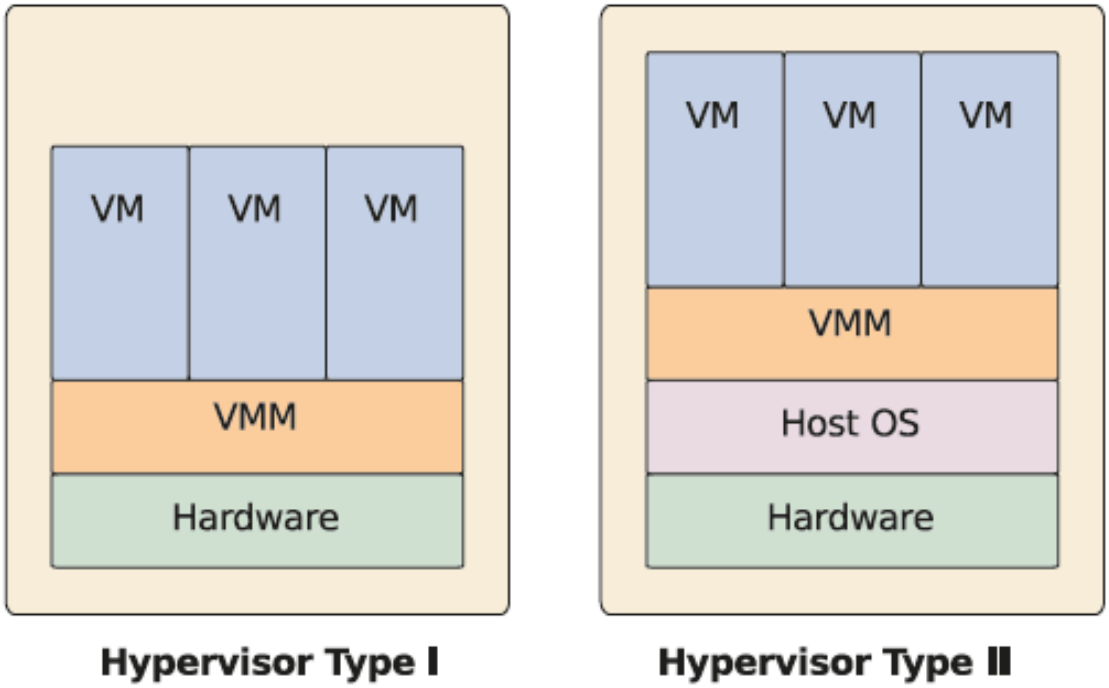

Figure 3.1 *Hypervisor types. [32]*

3.1.1 Challenges in x86 architecture virtualization

Some processor instructions, when executed in a virtual machine can not be executed directly on the host processor, as they would interfere with the state of underlying hypervisor or host OS. These kind of instructions are called sensitive instructions. The key to implementing a hypervisor is preventing direct execution of sensitive instructions. In the x86 processor architecture some instructions are privileged. This means they assume to be executed at the most privileged domain, otherwise they will cause an exception. Usually the hypervisor is being run in privileged mode and virtual machines in user mode, and when a privileged instruction is executed in a virtual machine, the exception will be trapped by the hypervisor. Hypervisor will then execute the instruction with proper behaviour. However, the x86 architecture has a total of 17 instructions which are sensitive but not privileged [31]. The processor will execute these instructions without generating an interrupt or exception, so they cannot be trapped and passed to the hypervisor. Theorem by Popek et al. [27] states; "for any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions." This requirement is not fulfilled by the x86

architecture by default, and for a long time the x86 was considered unvirtualizable. Nowadays, there are multiple solutions to the problem of x86 architecture virtualization; software-based full virtualization, OS assisted virtualization also known as paravirtualization and hardware assisted virtualization.[31, 27, 39]

3.1.2 Software-based full virtualization

Full virtualization done purely in software requires all the code run by the guest to be analyzed and transformed into safe instructions before executing. This approach is very costly in terms of performance and complexity. An example of a pure software-based virtualization technique is binary translation, where the guest OS kernel code is translated to replace the sensitive and non-privileged i.e. nonvirtualizable instructions with new sequences producing intended effects on hardware, while user level code is executed directly on the processor. [39]. This technique fully abstracts the guest OS from the underlying HW, which allows unmodified guest operating systems to be run. However, in binary translation, the guest operating system must be supported by the host CPU, so the support of unmodified guest OSs is limited to those that run on the host CPU. [32] Figure 3.2 depicts the binary translation approach.

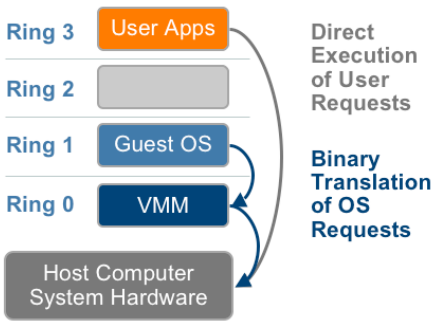

Figure 3.2 *The binary translation approach to x86 virtualization. [39]*

3.1.3 Operating-system-assisted virtualization

In operating-system-assisted virtualization or paravirtualization, the nonvirtualizable instructions that were trapped and translated in the purely software-based approach, are replaced by hypercalls in the guest OS kernel code. These hypercalls communicate with the virtualization layer instead of the host hardware. This can

offer some performance benefits compared to binary translation, but it is highly dependent on the workload [39]. As this approach requires modifications to guest OS kernel, it does not allow the use of unmodified operating systems. The OS-assisted method is shown in figure 3.3.

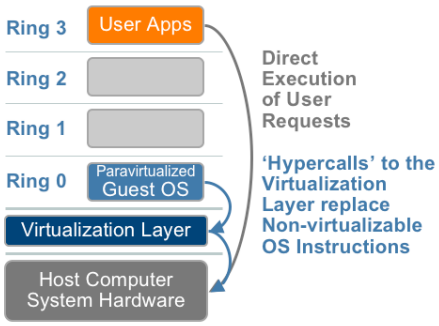

Figure 3.3 *The OS-assisted approach to x86 virtualization. [39]*

3.1.4 Hardware-assisted virtualization

As the third solution to x86 architecture virtualization, two main processor vendors Intel and AMD developed hardware-assisted virtualization technologies named VT-x and AMD-V, respectively. These extend the CPU architecture with virtualization support, which allows hypervisor to be run on a Root Mode, a privilege level below Ring 0. Hence, unmodified guest operating systems can be run on Ring 0, while sensitive and privileged calls are set to automatically trap to the hypervisor. This approach has advantages in terms of performance as all the trapping is done in hardware instead of software, and it is the preferred way to run unmodified guest OSs. In figure 3.4, the HW-assisted method is illustrated.

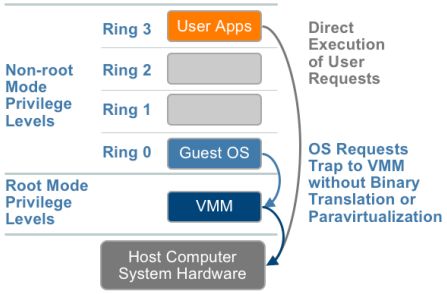

Figure 3.4 *The HW-assisted approach to x86 virtualization. [39]*

3.2 Containerization

It is not always necessary to create virtual machines running their own operating systems, when creating isolated environments within the host operating system is sufficient for achieving the benefits of virtualization. This is called containerization and it is virtualization on an operating system level, using jails or containers to isolate environments as processes in host OS. The idea behind containers is basically the same as in virtual machines, to create sandboxed environments that are using shared resources. All the containers in the host machine share the single host OS kernel, hence containerization is also called Shared Kernel Virtualization. Earliest example of a functionality related to operating system level isolation is chroot command introduced in FreeBSD OS in 1982[37]. Containers as a concept have been around from year 2000 as jails [36] on Unix operating systems, but in recent years the adoption of container-technology has grown exponentially [28].

Containers can be used for different purposes, and the size and structure of a container varies between use-cases. A container can run a single or multiple processes and services or even a full operating system. Containers that run multiple services or full operating systems are called system containers. Early implementations of containers were system containers like FreeBSD Jails [36] and Linux Vserver [18] released in 2000 and 2003, respectively. System containers can be thought of as lightweight VMs using shared kernel of the host OS, allowing better efficiency and smaller overhead.

An application container is a newer concept and it is designed to run a single process or service to embrace modularity and isolation. It includes everything needed to run that single application, so it behaves the same regardless of the execution environment. Docker, released in 2013, was the first application container technology and is the main reason to the exponential growth of the container-adoption in recent years. Docker was first to offer a full ecosystem for running and managing containers, including an efficient, layered container image model. The single process approach differentiated Docker from the traditional VMs and system containers, and it was easier to adopt and take advantage of by software developers.

3.2.1 Linux containers

Modern container technologies are largely based on Linux components, mostly provided by the kernel. A container is a fully isolated environment within a system. Isolation is provided by namespaces, a feature in Linux kernel which ”enables creating an abstraction of a particular global system resource and make it appear as

a separated instance to processes within a namespace"[29]. There are 7 different namespaces in Linux kernel; IPC, Network, Mount, PID, User, UTS and Cgroup. IPC or Interprocess communication is used to control the message queues a process can see. Mount -namespace allows creating isolated mount points to the file system and network- and user namespaces are used for isolating respective resources. Process visibilities are managed by the PID namespace and UTS controls the hostname and domain information.

Container resources are managed by a kernel feature called Control Groups or cgroups, which allocate e.g. CPU time, memory and network bandwidth among groups of tasks. Cgroup is the latest addition to Linux namespaces, and it is responsible for handling information about Control Groups. Security-Enhanced Linux or SELinux is a Linux kernel security module and can be used to enhance the security, namely access control policies of containers. There are also other security modules for Linux kernel, e.g. AppArmor, which is used in Ubuntu Linux distribution. Figure 3.5 illustrates the Linux container architecture.

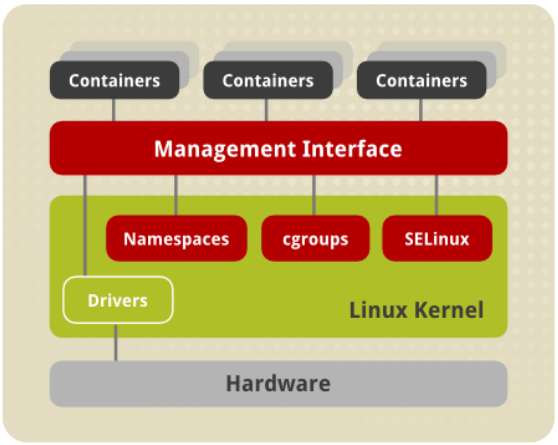

Figure 3.5 *The architecture of a RedHat Linux container. [29]*

The first container manager based on namespaces and cgroups was Linux Containers or LXC, released in 2008. Most of the modern container managers, including Docker, are using namespaces and cgroups in providing isolation.

3.2.2 Containers on Windows and macOS

As containers are based on Linux kernel components, running them in platforms other than Linux requires additional software layers. For Windows, there are currently three main solutions to run containers, each a bit different depending on the

version of Windows used as a host. All of these are leveraging the Docker Engine, as Microsoft has partnered up with Docker to bring the benefits of docker platform to Windows [17].

Linux containers are native containers, meaning they all run as processes directly on top of the host OS. In Windows, native containers are only available on Windows Server 2016 and newer server versions. Native containers are not available in earlier versions because of the limited amount of user modes. Earlier Windows versions have only one User Mode and one Kernel Mode. Each container needs to run in its own User Mode to provide isolation and resource governance, so availability for multiple user modes known as Container User Modes had to be introduced. The main user mode or Host User Mode runs all Windows core services and processes and also container management technologies, namely Compute Services. Compute Services API offers functionality for launching, restarting and monitoring the state of the containers. The API is used by Docker Engine, which handles all the container management.[42]

Second type of containerization on Windows is based on Microsofts Hyper-V[19] virtualization technology. Hyper-V is Microsofts integrated hypervisor solution and it was introduced with Windows Server 2008 and has been available on all server releases since. In Windows client operating systems Hyper-V has been supported since Windows 8, but only for the professional and enterprise versions. Containerization on Hyper-V is based on using optimized lightweight virtual machines to run containers. Every container is running in it's own VM with a dedicated kernel and a guest Compute Service interface for managing the state of containers. Hyper-V container is like a hybrid of VM and native container approaches, standing somewhere in between of the two when considering the overhead of virtualization. With LinuxKit[2] it is also possible to run Linux containers on top of Hyper-V.
Figure 3.6 shows a high-level architecture of a Hyper-V container.

On Windows 7 and versions before, containers can be run using Docker Toolbox, which is a set of tools for container hosting and management. In addition to all the Docker functionalities, it comes with a VirtualBox hypervisor, a lightweight Linux VM called boot2docker and some binaries for managing the virtual machine. As Docker Toolbox uses a Linux VM to host the containers, one can only run Linux containers with it.

On MacOS, containers can be run within a VM using the Docker Toolbox like in Windows. Another option is Docker for Mac [4], which uses a toolkit for hypervisor

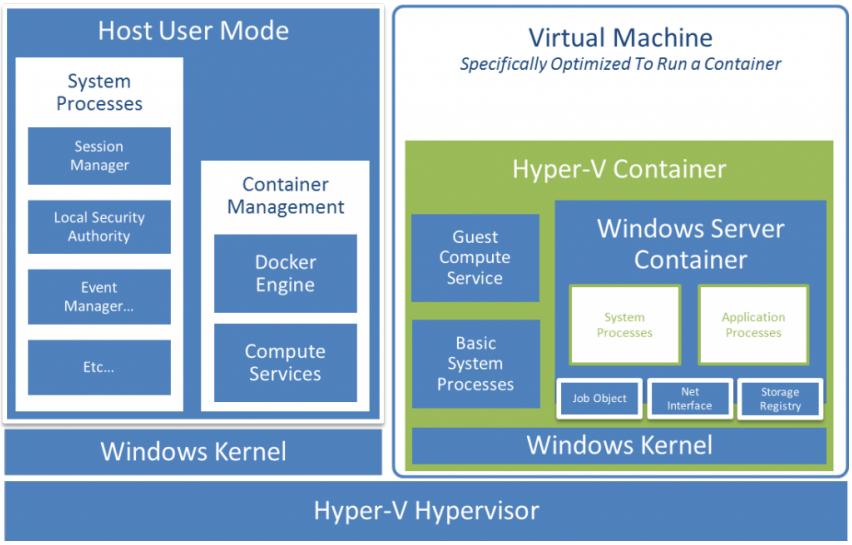

Figure 3.6 *The architecture of a Windows Hyper-V container. [43]*

capabilities called HyperKit. HyperKit includes a hypervisor based on xhyve, a lightweight virtualization solution for macOS. Docker for Mac is a solution similar to Hyper-V containers, using a dedicated VM as a Docker host. As of today, there are no native containers for MacOS since a VM is needed for both options for running containers. [21, 20]

3.2.3 Containers vs VMs

The main advantages of containers versus virtual machines are smaller size and virtualization overhead, faster deployment and greater modularity. As there is no need for a separate OS for each instance and even single processes can be containerized, containers use far fewer resources than virtual machines. Considering overhead in terms of memory usage, containers are more efficient because they can dynamically use the shared system memory, whereas virtual machines usually have their own preallocated memory and storage space. Also, in VM approach there are lots of duplicate processes running inside nested OSs, which can be avoided with containers. The speed advantage in deployment is obvious when comparing starting a process within a container versus first starting a full operating system in VM and then a process inside the OS. With containers it is possible to run just the one process or service needed, which offers a big advantage in modularity compared to VMs.

There are also some limitations and drawbacks on using containers. As the host OS kernel is shared, the operating system of all the containers must be compatible with the kernel. For this reason, one cannot run Windows in a container on Linux host

and vice versa, which is possible with VMs. There are techniques to bypass this problem, like running a VM as a container host or running a minimal linux kernel and userland on Windows to run Linux containers. The first approach obviously removes lot of the benefits of using containers as you need the virtual machine with all the added overhead. Another drawback in containers is the level of isolation. Since the kernel is shared, the unwanted effects have a greater chance of leaking into the host OS and to another containers. Also, when there are problems in kernel caused by a fault in a container, all the containers are affected. Virtual machines having their own kernels, only the specific VM is compromised in case of a kernel failure.

4. NIX - A FUNCTIONAL PACKAGE MANAGER

Nix is a package manager introduced by Eelco Dolstra in his thesis in 2006 [9]. The basic idea of Nix is to isolate software components in a central store, using cryptographic hashes derived from all the build-inputs of a component as path names. The approach is based on a purely functional model, which means that all the packages are built by functions whose outputs are only dependent on the inputs, i.e. function arguments. The functional model has many advantages in package management. In traditional package management systems, there are a lot of problems which Nix can solve; inexact dependency specifications, non-coexistense of multiple versions of a software package, interference between software components, unatomic upgrades, etc. Next sections take a closer look on Nix and how its design helps to avoid these problems. [10]

4.1 The Nix Expression language

Nix uses its own language called the Nix expression language. It is a simple, untyped, purely functional language designed only for describing components and their compositions. It is not meant to serve as a general purpose programming language. Like all functional languages, it uses a declarative paradigm, meaning programming is done with declarations or expressions instead of statements. State-changing and mutable data are avoided. Nix expression language is pure, which means everything is produced by functions, and the output of functions depend only on the input arguments. This leads to side-effect free, reproducible builds. Another feature in Nix common to other functional languages is laziness, which is a term used to describe a language that evaluates expressions only when needed for an output. All these characteristics work well with package management, where consistency and reproducibility are important. [9, p. 61-86]

4.2 Nix store

The central store where all the components are installed is called a Nix store, which is simply a designated directory in the file system. Components are directory trees

containing subdirectories like `/bin` and `/lib` for component-specific files. Everything in Nix store is marked as read-only, so components never change after they have been built. Path names of components consist of a 160 bit fixed length hash followed by the component name and version, e.g. **/nix/store/bwacc7a5c5n3qx37nz5drwcg-d2lv89w6-hello-2.1.1**. The hash is computed from all the inputs to the build process of a component. Hashes are collision-resistant, meaning it is not possible for two different inputs to produce the same hash.[9, p. 19-24]

4.3 Nix expressions

Components are built using Nix expressions, which are functions to describe components and their dependencies. Deploying a component with Nix requires three parts: a Nix expression i.e. function describing the component with all the inputs, a builder script to build the component from the inputs, and a composer expression to call the function with appropriate arguments. Below is a simple Nix expression for building an example application named hello [23].

```
{ stdenv, fetchurl, perl }:

stdenv.mkDerivation {
  name = "hello-2.1.1";
  builder = ./builder.sh;
  src = fetchurl {
    url = ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
    sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
  };
  inherit perl;
}
```

Nix functions are formed using syntax: { `x, y, z` }: e, where `x,y,z` are function inputs and `e` is the body of the function. So the first line states that the expression is a function expecting three arguments: `stdenv`, `fetchurl` and `perl`. The first argument, `stdenv`, is a package that provides a basic Unix-like environment with tools like gcc, bash, cp, grep etc. It is used in most of the Nix packages. `fetchurl` is a function to download files and `perl` is a Perl interpreter. Build actions in Nix are called derivations, which usually consist of a build script, a set of environment variables and a set of dependencies [10]. Looking at the example, `stdenv` provides a function called `mkDerivation`, which builds the package from a set of attributes:

`name`, `builder`, `src` and `perl`. Attributes are key-value pairs where key is a string and value is an arbitrary Nix expression. The inherit command does essentially the same as defining an attribute `perl = perl`, just in a more concise form. All the attributes are passed in to the builder as environment variables. A builder script for the example application is described below [23].

```
source $stdenv/setup

PATH=$perl/bin:$PATH

tar xvfz $src
cd hello-*
./configure --prefix=$out
make
make install
```

In the builder script, the attributes `src` and `perl` defined in the earlier Nix expression are used as variables. Attribute `stdenv` points to the location of the standard environment and `mkDerivation` -function adds it automatically. The setup script from the `stdenv` is always called first. It cleans the environment to prevent unwanted inputs from being used in the build. Next, the `PATH` is set for the required binaries, in this case for `Perl`. Finally, the sources are unpacked, configured and built using make. Nix creates a hash from all the derivation attributes and passes it to the builder as an environment variable called `out`. This variable is used as a prefix for the installation path defined in configure -script.

The last part needed when building a Nix component is the composer expression used to call the Nix expression with correct arguments. Usually the composing is done in a file called all-packages.nix, which is a large file containing imports and function calls for all the packages in a Nix system. Following is an example of composing the hello application in all-packages.nix [23].

```
rec {

  hello = import ../applications/misc/hello/ex-1 {
    inherit fetchurl stdenv perl;
  };
  perl = import ../development/interpreters/perl {
    inherit fetchurl stdenv;
  };
  fetchurl = import ../build-support/fetchurl {
    inherit stdenv; ...
  };

  stdenv = ...;

}
```

The rec in the first line means that a mutually recursive set of attributes are going to be defined, meaning the attributes can refer to each other. Importing reads and returns the Nix expression from the defined path, and the imported expression is called with the arguments inside the curly braces. [9, 10]

4.4 User Environments

Nix handles user-specific configurations through user environments, which are based on directory trees created from symbolic links. A user environment is a separate component in the Nix store which includes symbolic links to software components activated for the user. Whenever a user performs Nix operations that alter the environment, a new version of the user environment, called a generation, is created. Generation is simply a symbolic link to a specific user environment in the store. Generations are further grouped into user-specific profiles. User environments enable atomic upgrades, which in general means that upgrade is either fully done or nothing has changed. As upgrading packages in Nix always creates a new generation without altering the current one, and the environment is changed to point to a new generation by an atomic symlink -operation, upgrades are atomic. This design makes also rollbacks to previous configurations simple and atomic. User environments are illustrated in figure 4.1. [9, p. 36-38]

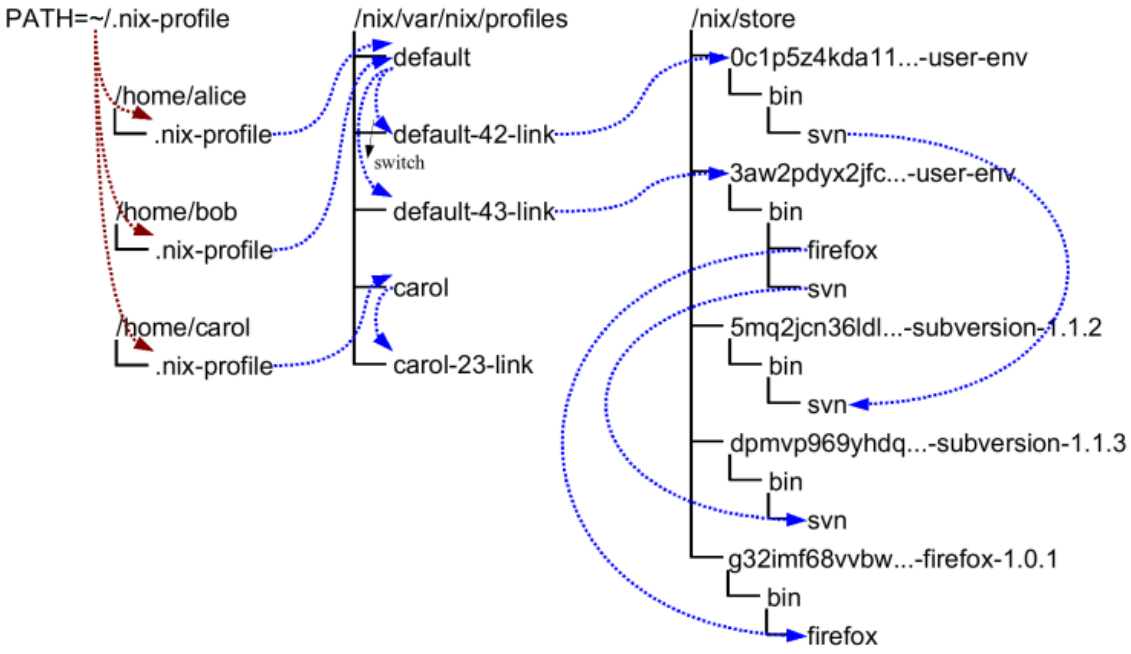

Figure 4.1 *Nix user environments. [23]*

4.5 Garbage collection

Removing components in Nix is never handled by the user, only the garbage collector can remove components. Nix keeps track of all the user environments in the system, and the garbage collector can only remove packages that are not part of any user environment. Since Nix inherently provides accurate information about dependencies and garbage collector only removes independent packages, it is always safe to execute garbage collection. The garbage collector does not run automatically, it has to be instantiated by the user. [9, p. 38]

5. DIFFERENT APPROACHES TO ISOLATION

5.1 Isolation by virtualization

5.1.1 A development VM

A virtual machine is running in its own virtualized hardware and operating system, so it naturally provides full isolation from the host and other projects. The idea behind using VMs for development is that each development environment has its own virtual machine. All the necessary tools and software components are installed in this development VM. It is used only for one project, and nothing environment-specific should be configured or installed manually, unless it is for testing purposes and the VM will be recreated after testing. Whenever there is a permanent configuration change needed, it should be added to the provisioning scripts that are run when the VM is booted for the first time. This way the environment stays in a known state and is always reproducible.

There are options on how to use a VM for development, and which approach to choose depends on the project. One can run an OS without a GUI, meaning no graphical user interface installed, keeping the OS image smaller. Usually one will need graphical tools like IDEs or browsers in development, so running the development VM without a GUI may require configuring a shared folder and port mappings between the VM and the host. In this setup, building and execution of the application is performed by the VM, host is used for editing the source code and viewing the state of the application. This way of developing partially breaks the reproducibility of the environment, because it requires tools from outside the isolated environment.

Another option is to run a full OS with a GUI in the development VM. This way one can install all the tools, including IDEs, browsers and version control systems to the VM. There is no need for a shared folder or port mappings and all the tools and versions are synchronized across all the developers. Running everything from the VM has some obvious drawbacks, mainly non-native experience which presents itself as slightly slower response times, graphical performance problems and other performance issues due to limited resources. Another feature on this setup

is that all the developers have to use the same toolchain. This can be viewed as a good thing as there is no disparity between tools, but on the other hand developers might have different preferences. These are some of the trade-offs between isolation, performance and preference, and it depends on the project and people what is acceptable.

Usually dividing services across multiple VMs in a development setup is not a very viable option because of the huge resource overhead. There are some scenarios where one might have to use more than one VM for development, for example when different operating systems are needed for some services.

5.1.2 Containerized development environment

Isolation provided by containers is achieved by namespaces, like discussed in Chapter 3. All of the namespaces provide a different layer of isolation. With mount-namespace one can isolate a filesystem and with a network-namespace one can create an isolated network [35]. A containerized development environment from a user perspective is a lot like a development VM with the shared folder approach. Source code is edited on the host and shared with containers. The main difference is that the environment is usually composed of multiple containers running a single service instead of running all the services in the same filesystem and namespace. As there is no one common filesystem, one or more container-volumes to share data between containers may be needed. All the application-related configurations and software installations should be done in containers, nothing should be configured locally. The project naturally dictates the final form of the environment, but the main idea is to keep the host machine as clean as possible to ensure reproducibility.

5.2 Isolation with Nix

The main property in Nix that allows creating environments with isolated software components is the hashing. As discussed in the previous Chapter, Nix uses hashes computed from all the build inputs in component paths. This forces installed or upgraded components to always produce a new path to the Nix store. The old versions of components are kept, so one environment can change to the upgraded package while others can continue to use the old one. Component isolation provided by Nix does not prevent multiple environments from using the same package, it just ensures that changes only affect the target environment.

Creating a development environment with Nix is based on creating a dedicated,

preconfigured shell user environment including only the packages needed for development. The environment is described in a derivation -file and instantiated with Nix. There is no namespace isolation involved, the isolation provided by Nix is purely based on its package management model and the configured shell session.

6. PRACTICAL EXAMPLE - ISOLATING A JAVASCRIPT WEB DEVELOPMENT ENVIRONMENT

In this chapter, three solutions for isolating a web development environment are presented. The idea is to provide some insight on how to use virtual machines, containers and Nix with some auxiliary tools to set up a development environment. Only the high-level configurations like installations and basic setups are covered. Detailed workflows and configurations are very project-specific, so they are left outside the scope of this thesis.

6.1 The environment

The environment under examination is a typical JavaScript development stack using MongoDB database and Node.js runtime. The environment includes components like React JavaScript library, Mocha test framework, task runner gulp and code analysis tool ESLint. They are all part of the node ecosystem and installed with npm, thus having no real impact on setting up the environment. One thing to note is that Node.js environments are not necessarily the best candidates for the heaviest isolation solutions, because they can be pretty self-contained when used right. The focus is on how to achieve the isolation using the tools presented, not in the details of the environment.

6.2 VM approach

In this section the JavaScript development environment is isolated using a virtual machine. The example presented here requires two software tools; VirtualBox and Vagrant. These tools are briefly introduced first before presenting the actual steps needed for setting up the environment. Installation of tools is not included in the steps.

6.2.1 VirtualBox -hypervisor

One of the most popular hosted hypervisors for running VMs is Oracles VirtualBox. It supports all major operating systems, including Linux, Windows, macOS and Solaris. VirtualBox is capable of using all of the main x86 virtualization techniques. However, to run 64-bit architectures which most OSs today are using, VirtualBox requires hardware-assisted virtualization. Support for this is available on most of the modern CPUs. [25, p. 250-255]

VirtualBox has a feature for sharing a host folder with the virtual machine. The feature is provided by VirtualBox Guest Additions [26]. The guest OS has read and write access to the files in the shared folder by default, but the access policies can be modified. The implementation of the shared folder varies with an OS; in Linux it is a virtual file system and on Windows it is a pseudo-network redirector. [25, p. 71]

6.2.2 Vagrant for managing the environment

Vagrant is an open source software for managing virtual machines and more specifically, development environments. Vagrant offers a convenient system around virtualization providers, wrapping the functionality with a declarative syntax and simple commands. Its main function is to start and provision a virtual machine by configuring the VM and installing and configuring all the software tools needed for the environment. Provisioning means configurations that are done after the VM has been started. Figure 6.1 shows the basic workflow of Vagrant. There are multiple provisioners available, including shell scripts, Ansible, Docker, Chef and Puppet. Provisioners can be run from the host machine, or installed to and used from the VM. Vagrant also sets up shared folders and ports between the VM and the host for development. In addition to VirtualBox, Vagrant supports other virtualization providers like Hyper-V, VMWare and Docker. Vagrant with VMWare is not free of charge, while other options are.

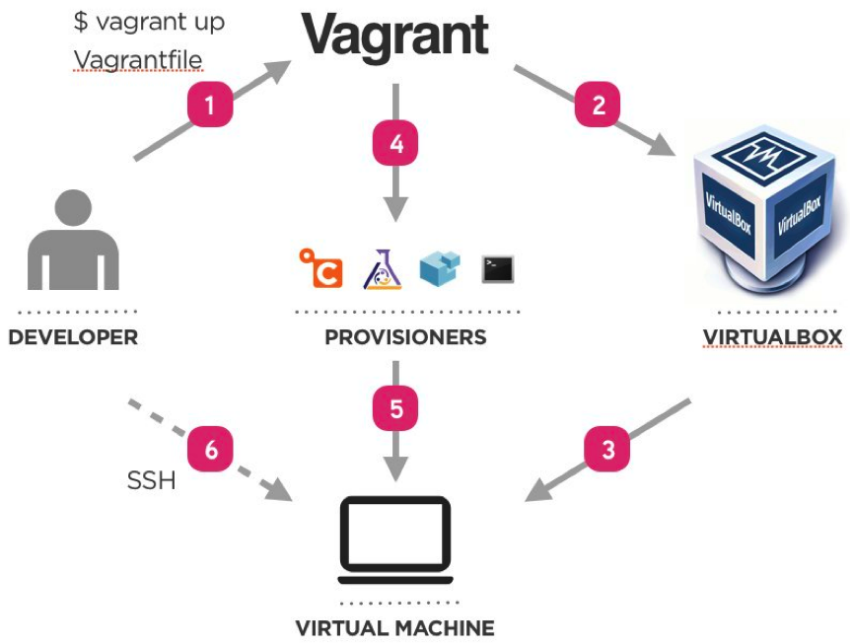

Figure 6.1 *Basic Vagrant workflow. [23]*

Vagrant VMs are called boxes, which are essentially virtual machines with some added tools, configurations and metadata. A basic Vagrant box or base box as they are called, typically contains at least a package manager and SSH with a user created for Vagrant. Also some other configurations may be added depending on the OS and virtualization provider used. VirtualBox needs VirtualBox Guest Additions installed on the VM in order to use synced folders between the host and the VM. Base boxes can be uploaded to and used from Vagrant cloud, a repository for Vagrant boxes. One can also customize their own VM manually and then create a box from it with Vagrant. [15]

With Vagrant, the whole development environment is configured in one file called Vagrantfile. This file is committed to version control with source codes and other configurations. Vagrantfile for the JavaScript environment used is presented below.

```
Vagrant.configure("2") do |config|

  config.vm.box = "debian/jessie64"
  config.vm.box_version = "8.10.0"
  config.ssh.username = "vagrant"

  config.vm.network "forwarded_port", guest: 27017, host_ip:"127.0.0.1",
host: 27017, id: "mongodb"
```

```
  config.vm.network "forwarded_port", guest: 3000, host_ip:"127.0.0.1",
host: 3000, id: "node.js"

  config.vm.synced_folder ".", "/vagrant", type: "virtualbox"

  config.vm.provider "virtualbox" do |vb|
    vb.name = "debian-dev"
    vb.gui = false
    vb.memory = "2048"
    vb.cpus = 4
  end

  config.vm.provision "shell", path: "provision/provision.sh"

end
```

A basic Vagrantfile like above is pretty self-explanatory, but let's elaborate on some of the steps. Here the chosen box is 64-bit Debian release version 8, also known as jessie. The size of the box is 277 megabytes in a packaged format, and 940 megabytes unpacked. Depending on the project and environment requirements, one could use a smaller Linux distribution like Alpine Linux as well. The box is searched and downloaded from the Vagrant Cloud by default. One can also use a direct URL to a Vagrant box, or define a local path for the used box. The forwarded ports and synced folders are defined for the development environment in respective lines. Next, the VM is named and allocated a certain amount of memory and CPUs. Graphical user interface is disabled because there is no GUI in the chosen Linux box and the command-line can be used through SSH from a developers shell instead of the VirtualBox GUI. Finally a provisioning script is launched after the VM has booted. The provisioning is done using a shell script, which is fine in a simple setup like this one. In more advanced environments with lots of configurations, a provisioner like Ansible might be useful. The provisioning script used is presented below.

```
#!/bin/bash

# Import key for mongodb
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2930ADAE8
CAF5059EE73BB4B58712A2291FA4AD5

# Create a list file for MongoDB
```

```
echo "deb http://repo.mongodb.org/apt/debian jessie/mongodb-org/3.6 main"
| tee /etc/apt/sources.list.d/mongodb-org-3.6.list

# Install MongoDB
apt-get install -y mongodb-org=3.6.4

# Start MongoDB service
service mongod enable
service mongod start

# Install nodejs
apt-get install -y curl
curl -sL https://deb.nodesource.com/setup_8.x | bash
apt-get install -y nodejs

# Install fontconfig for PhantomJS
apt-get install -y fontconfig

# Install node_modules
mkdir -p /app; cd /vagrant/app; cp package.json package-lock.json gulpfile.js
/app
cd /app; npm install --unsafe-perm=true --allow-root

# Create a symbolic link for shared folder node_modules
ln -sf /app/node_modules /vagrant/app/node_modules

# Add node_module binaries to path
echo "export PATH=$PATH:/app/node_modules/.bin/" >> /home/vagrant/.bashrc

# Set current directory to app directory by default
echo "cd /vagrant/app/" >> /home/vagrant/.bashrc
```

The provisioning script is fairly straight-forward. A key for MongoDB is imported and a list file is created first. Next, software packages for MongoDB are installed and MongoDB -daemon is started. Then, nodejs and fontconfig for PhantomJS are installed. The next two steps make sure that node modules are not placed on the host machine through the shared folder. Instead, node modules are installed in a different folder in the VM from where a symbolic link is created to the shared folder. This is done to keep the host machine as clean as possible, and also to avoid problems

with symbolic links with npm and the shared folder. The folder that contains node module binaries is added to user path to be able to use node binaries directly from the command-line. Finally, for convenience, default directory is set to the directory containing the application.

The presented Vagrantfile and provisioning script describes all the configurations for setting up a simple nodejs development environment. The VM is instantiated with a command `vagrant up`, run from the folder where the Vagrantfile is located. While the VM is running, a developer is able connect to it with SSH by executing command `vagrant ssh`. All changes to the source codes in the shared folder are being reflected to the development VM.

6.3 Container approach

In this section, the JavaScript web development environment is containerized. Docker is chosen as the container platform, mainly because of its wide-spread and efficient ecosystem. First, a quick overview on Docker and its features is presented, after which the steps for containerization are discussed.

6.3.1 Docker containers

Docker leverages cgroups and namespaces when creating containers, like discussed in Chapter 3. The control groups are used to allocate resources for containers and namespaces provide isolation. The namespaces used by Docker are listed below [5]:

- pid: Process isolation (PID: Process ID).
- net: Managing network interfaces (NET: Networking).
- ipc: Managing access to IPC resources (IPC: InterProcess Communication).
- mnt: Managing filesystem mount points (MNT: Mount).
- uts: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

The file system Docker uses is UnionFS or union file system. It is a system based on copy-on-write, a resource-management technique where resources are never edited but instead a new copy is always created when writing data. It enables efficient sharing and reduces the amount of I/O operations. Docker can use multiple flavors

of UnionFS, depending on the operating system's support. The default container format for Docker is libcontainer, which is a wrapper format combined from control groups, namespaces and the union file system. [5]

A Docker image consists of a series of layers, where each layer represents an instruction in a file called Dockerfile. Every Dockerfile starts with a FROM command, which defines the base image on top of which the container is built. Each subsequent line creates a new layer. Some of the basic instructions create a layer with a size of zero. These layers can be overwritten when the container is started. The last layer always specifies the command to be run when the container is started and it is also overridable. Dockerfile for the node environment under inspection is presented below.

```
FROM debian:jessie

RUN apt-get update -y && apt-get install -y curl bzip2
RUN curl -sL https://deb.nodesource.com/setup_8.x | bash && apt-get
install -y nodejs libfontconfig

WORKDIR /app
COPY package.json package-lock.json gulpfile.js /app/
RUN cd /app && npm install
ENV PATH="$PATH:/app/node_modules/.bin/"

COPY . /app/

EXPOSE 3000

CMD bash
```

The image used is from Docker Hub, an official registry for Docker images. It is a container image based on the same Debian release that was used with Vagrant. This image is 127 megabytes in size. There are smaller images for running node applications, but this was chosen because it was able to run PhantomJS that is required in some of the tests for the environment. Some additional tools like `curl` and `bzip2` are installed to be able to use default installation methods of node and PhantomJS. In `COPY` -command, files that have all the node dependencies listed are copied to the working directory of the container, from which npm install is executed. The reason these files are copied separately in an earlier step is that dependencies would not get

updated in the next build without it. If everything is copied after installing dependencies, subsequent builds would use Dockers cache and new dependencies would not get installed. The PATH is set because it allows node modules like `gulp` to be run directly from the command line, without installing anything globally. Finally, everything from the current folder is copied to the container, port 3000 is opened for the container and the startup command is defined. As this is a development container, a bash shell is started. For a production -container, the CMD would be something that starts the application. Figure 6.2 illustrates the layers and their sizes in the node development container.

dc15158d1f14	CMD bash	0 B
d2f908be4168	EXPOSE 3000	0 B
c3f0b50c67d0	COPY . /app	16.5 MB
3c777b81603b	ENV PATH="$PATH:/app/node_mo..	0 B
2d7621d1106c	RUN cd /app && npm install	520 MB
18e61ab44979	COPY package.json package-lo..	591 KB
67d36d33cf35	WORKDIR /app	0 B
b8393d87f645	RUN curl -sL https://deb.nod..	105 MB
f3b384d41396	RUN apt-get update -y && apt..	24 MB
	base image: debian:jessie	127 MB

Figure 6.2 *Layered structure of the node development container.*

6.3.2 Multi-container environment with Docker Compose

Docker Compose is a tool for describing and running Docker applications consisting multiple services. The application is defined in a docker-compose YAML-file, which includes top-level configurations for all the services in the application [6]. This file can be used to describe a development environment with all the containers needed. Below is an example of a `docker-compose.yml` for the environment.

```
version: "2"
```

```
services:
  web:
    build: .
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - /app/node_modules
    command: gulp
    depends_on:
      - mongodb
  mongodb:
    image: mongo
    volumes:
      - ./data/mongodb/db:/data/db
    ports:
      - "27017:27017"
```

There are two services defined; web and mongodb. Container for MongoDB is available in Docker Hub, and as it is defined as the used image, it will be downloaded when starting up the environment for the first time. Web is the node application, built based on the Dockerfile presented earlier. In ports -section, the default port in node applications, 3000, is mapped from the container to the host. This mapping allows the application to be used from the hosts browser. The first volume definition is mapping the source folder to the container, so code can be edited locally while application is running in the container. Because node modules are installed under `/app` -directory in the container, mapping the source folder to `/app` will hide the node modules. The second definition creates another volume to path `/app/node_modules` inside the container, which effectively unhides the node modules. There are also other ways to handle the problem with the hidden node modules, for example by using symbolic links like with Vagrant in section 6.2.2.

Both containers are instantiated with a command `docker-compose up`, run from the folder where the `docker-compose.yml` is located. MongoDB -container will start up first, because it is defined as a dependency for the web -container. While the containers are running, the node application will be available on port 3000 on the host, and all changes to the source code are reflected to the running application.

6.4 Using Nix

This section describes how the JavaScript environment is isolated using Nix. As mentioned in section 5.2, Nix provides isolation on a software component level. Installation of Nix is the only requirement for this setup. The preferred installation method is a simple shell script offered by Nix, which can be redirected to a shell [23]. An important thing to note is that in this example Nix is used only to provide isolation for Node.js and MongoDB components, so that multiple versions of those could be installed and used simultaneously without interference. Node can handle its own dependencies without interference as long as packages are not installed globally. It is possible to use Nix to deploy node projects with a tool called `node2nix` [34], but in this example the original node workflow is retained and npm is used to install node dependencies.

6.4.1 Isolated environment with a Nix Shell

Nix Shell provides a functionality to either build from source or download a set of packages defined by the user and start a shell session having the defined packages available with all the necessary environment variables set. When `nix-shell` is started with a flag `--pure`, the environment is cleared from some of the existing variables like PATH, so no external packages can be used by accident. However, Nix shell is not actually fully pure even with the flag, because it still leaves some things from the user environment uncleared. This impurity and isolation level will be discussed more in Chapter 7.

The Nix expression language is quite flexible, so there are lots of different ways to define things in Nix. The solution presented here uses an expression file and Nix Shell -functionality to setup the environment. The idea is that expression file describes the development environment fully, and it will be committed into version control with source codes. Running a command `nix-shell` from the folder where the expression file, called `shell.nix` is located, is everything needed to start up the environment. The expression file for the node environment is presented next.

```
{
  pkgs ? import (fetchTarball {
    url = https://github.com/NixOS/nixpkgs-channels/archive/00e56fbbe
    e06088bf3bf82169032f5f5778588b7.tar.gz;
    sha256 = "15pl5p3f14rw477qg316gjp9mqpvrr6501hqv3f50fzlcxn9d1b4";
```

```
  }) {}
}:
  with pkgs;
  stdenv.mkDerivation {
    name = "node-env";
    buildInputs = [ nodejs-8_x mongodb ];
    shellHook = ''
      export PATH="$PATH:$(pwd)/node_modules/.bin/"
      alias mongod="mongod --dbpath $(pwd)/data/db --bind_ip $(pwd)/
      data/mongo.sock"
      npm install
    '';
  }
```

In the first step, a certain revision of Nixpkgs is downloaded and verified with a hash. It is important to pin down a specific version of Nixpkgs like above, otherwise, another developer running the expression later might get a newer revision with different packages. Next, a derivation is made from the downloaded package-set. Attribute `buildInputs` defines all the software packages needed and `shellHook` defines commands to be run after the shell has been started. The first command in the shell hook sets the binary folder of node modules to user path. This way all the modules can be used from the command-line without any global installations. The second line sets an alias for starting up a MongoDB daemon using socket binding. Finally, npm install is executed to get all the node dependencies. Commands in shell hook are just examples how one can customize the environment within the derivation.

This file used with Nix enables reproducible development environments with components installed in isolation. Installed node and mongodb packages will not conflict with anything else installed on the machine. As the revision of Nixpkgs is pinned down, the environment produced by nix-shell will always be the same. [23]

7. COMPARISON OF SOLUTIONS

In this chapter some of the specific features of different isolation methods presented are discussed. The features chosen are relevant to development environments. These features are level of isolation, reproducibility, resource overhead, usability and support and availability.

7.1 Level of isolation

A virtual machine generally provides the best isolation from the host and from other development environments. However, when using a shared folder approach discussed in section 5.1.1, where a folder is mapped from the host OS to the VM, the isolation is compromised. For example, the host OS and hypervisor used may have an affect on how the application on the VM behaves. One typical source of issues is symbolic links on shared folders, which have inconsistencies on some host/guest combinations [16], like Windows and Linux. Another thing that might cause issues on Windows hosts is that file permissions for all the files in the shared folder are set during boot-time, and cannot be changed from the VM.

Although containers are not as separated from the host as VMs and thus provide a weaker level of isolation, the isolation provided by containers is actually more useful when considering development. The difference is the scope of the isolation, which is much narrower when using containers. VMs isolate the guest operating system as a whole, as containers allow isolating single services or processes. Being able to isolate single services creates a more modular environment with a better separation of concerns. This can help in debugging issues or when testing different configurations. In a containerized Docker environment, there are similar issues with symbolic links as in VMs. On Windows hosts, symbolic links only work when created within containers.

Nix cannot really compete with virtualization in terms of isolation level. It only provides isolation for installed software components. There are situations when e.g. file system isolation and and network isolation will add benefits to the development workflow, and those are not possible with Nix. One benefit in an isolated file system

is that it's harder for the user or processes to execute harmful commands on the host OS. On Nix environment, the whole host OS file system is accessible and therefore vulnerable, while using containers or VMs only the shared folder is compromised. A scenario where network isolation would be beneficial is when running multiple versions of the same service simultaneously to do some comparison. Note that this is mainly feasible using containers, using multiple VMs would not be as practical because of the huge resource overhead and less flexible setup.

Nix shell environments have also another issue regarding isolation. When starting a nix-shell session, users shell startup file is read. As users can have anything in their startup file, this is a direct route to leak some undeclared dependencies or configurations to the environment. For example, one might have a path to a common binary set as an alias to `.bashrc`, which would then be available in the Nix shell. This is why Nix shell environments are not truly pure and side-effect free, even when using the `--pure` flag.

7.2 Reproducibility

Reproducibility is an important aspect of a development environment setup. An environment that might end up different based on the time of building or platform it was built on will lead to problems. With VMs and containers, there are two common things that cause issues in reproducibility. These are related to versioning and repositories, with both images and software packages. This section takes a closer look on those issues.

7.2.1 Problems with versioning

The way Docker image tags are used is not consistent. Docker Hub does not force any particular tagging convention, so there are a variety of tags available. Images in Dockerfiles might be referred only with a major version, e.g. node:8 or without a version tag at all. The obvious problem here is that depending on the time of execution, different images will be downloaded. There are situations where it is actually purposeful to always have the latest version, but when reproducibility is important, this should be avoided. Reproducibility in mind, the right way to use tags is to always use the most specific tag available, e.g. node:8.11.1. That being said, as there is no fixed convention on tagging and principles of semantic versioning [33] are rarely followed in Docker tags, there are no guarantees on getting the same image every time. When using only tags to refer to images, one will have to assess each situation separately to see if there is room for an image mismatch.

Even if one manages to use tags correctly, there is an inherent problem with reproducibility in the Docker ecosystem. Images referred by tags are updated in-place all the time. Constant security updates and patches are being run for the bottom level images like operating systems on top of which most of the images are built. Whenever there is a change in a parent image it will propagate to all of the descendant images at some point. So there is no real reproducibility on Docker ecosystem, unless every change is under one's own control. Full control would mean hosting ones own registry for Docker images.

What might seem like a solution for this problem in Docker is using a digest when referring to images [7]. Digest is a SHA256 hash, unique for each image. So instead of using tags, one can use the digest, e.g. `node@sha256:8c03bb07a531c53ad7d0f6e7041b64d81f99c6e493cb39abba56d956b40eacbc`. This is obviously not as user-friendly as using tags, especially when images pulled with a digest do not have tags at all, which makes it harder to recognize images. Looking at a Dockerfile using a hash to pull the image, one does not know what was the actual image used as a base. Adding a comment describing the tag would help with this, but the image still would not have a tag when listing images. Another problem with digests is that being unique, they change every time the image is updated. Using a common registry where images are being updated constantly, a build using a digest would fail often.

The situation is slightly better in VM world using Vagrant. Vagrant promotes semantic versioning and advocates the use of X.Y.Z -pattern for versions [14], although using semantic versioning is not compulsory. However, as VMs always use full operating system images, there is no nested images which leads to less variability and better reproducibility. When exact versions, e.g. 1.2.6, are always used instead of ranges, one should get the same image. However, as the images in Vagrant Cloud are managed by third parties, there are no real guarantees that the image will exist or has not been edited under the same version number.

7.2.2 Changing repositories

The second issue is related to software packages and repositories and it applies both to VMs and containers. Whenever a package is installed by a package manager in Vagrantfile or Dockerfile in Linux environment, the package is fetched from a repository. Most of the time repositories are preconfigured in the base image and may point to a repository with a constantly changing package set. On top of that, packages might be installed without a specific version number. It is self-evident that this will lead to disparity.

A reasonable first step to try to make things consistent is using specific version number when installing packages. This will make sure the same package is fetched every time, assuming the package exists or if there is not a newer minor version available. As repositories tend to change and many of them keep only one version of a package, the package one was able to install earlier might not exist or it might be different the next day. So pinning down packages is not enough, one would need to also pin down the repositories. This would require erasing the existing repositories from images and replacing them with a repository having a frozen package set. Even though some Linux distributions keep a wide collection of old repositories [1], it is not the case with most of the repositories providing packages.

At the moment using virtualization techniques, there is no easy solution for achieving true reproducibility when creating development environments. Keeping all the variables under control requires hosting and maintaining your own repositories for both images and software packages. This is a demanding task and not everyone has resources for that.

7.2.3 Nix and reproducibility

Nix obviously works very differently compared to VMs and containers. As discussed earlier in this section, the main sources of problems in reproducibility with virtualization were related to images and software repositories. Nix does not virtualize anything, so there is no dealing with images. Reproducibility is built in to the functional model. As the output artifacts are solely based on the function inputs, the environments created with Nix will always be the same. This naturally requires that the function inputs are the same. This is not the situation by default. Upon installation, Nix will be pinned to a certain revision of package repository. For another developer, a newer revision might be pinned. So pinning done the same version of the repository is paramount.

The Nix Packages collection is a repository for all the Nix packages. Like discussed in section 4.3, Nix packages are built using definition files: expressions, builders and composers. There are no binaries in the Nix packages repository, which makes it small in size and portable compared to binary package repositories. However, there is a separate cache repository for Nix packages, where binaries created from Nix expressions are stored. If the package defined in the expression can be found from the cache, it will be downloaded as a binary instead of compiling from the source. Whereas pinning down each revision of a software repository in traditional package systems is not viable because of the large size, it is possible in Nix ecosystem where simple functions are provided for this. The function fetchTarBall used in the

example in section 6.4.1, will fetch the specific version of Nix packages and use that as a base for derivations. This way the all the packages come from the same source and developers will get matching environments, regardless of when being built.

7.3 Resource overhead

Comparing resource usage, or more accurately, resource overhead of virtual machines, containers and Nix is not necessarily fair. They all provide different levels of isolation, which is more or less directly related to resource usage. Isolation provided by Nix is a result of its designed structure and not a product of virtualization that would cause overhead. Thus, it is considered a native solution. It is clear that native applications have better performance than virtualized ones, even though containers perform almost equal with native solutions on most situations [12]. The resource overhead discussed in this section is not however related to scalable cloud applications, it is about differences in size of the footprint of the development environment. In other words, how much resources the environment consumes to be able to provide its level of isolation. Even though the isolation levels are not equal, it is still useful to have a rough idea of how much resources each of the solutions waste. The purpose is not to get exact measurements, but only an approximation to be able to do some comparison. This section covers some examples on resource usages running the Node.js application. The categories that are being discussed are CPU-, memory- and disk usage. CPU and memory usage testing is done by running the unit test set of the node environment, which has over 2500 unit tests. Scenarios are run on a Lenovo IdeaPad 520S laptop with a Fedora 28 operating system.

7.3.1 CPU usage

The CPU usage was observed from the host machine by `top` process viewer while running a unit test set for the node environment. The run took about 1 minute and 16 seconds when the tests were run from a Docker container or from a Nix environment. The average CPU usage during the tests for both was around 12 %. Executed from the VM, the unit test set created a lot more varying workload for the CPU, ranging from 6 % to almost 30 %, and average being around 18 %. Also the execution time was more fluid and a bit longer for the VM, from 1 minute and 30 seconds to over 2 minutes. These simple test runs reinforce the notion that containers perform equally to native environments in terms of CPU, and VMs have some performance degradation due to added virtualization layers. It should be noted though that the results might diverge with a different kind of workload.

7.3.2 Memory usage

The main difference in memory usage between the solutions is that for VMs a fixed amount of memory is allocated at startup, while Docker containers and Nix can use system memory dynamically. This can be a problem when considering using VMs for development in a low-memory setup. Rarely all of the allocated memory is being used so VMs will waste system memory most of the time. The memory usage for the container development environment during the unit tests was ranging from 500 MB to 1000 MB, the average being at about 700 MB. For the development VM, the memory usage observed from inside the VM was averaging at about 900 MB. Hence, running a development VM with as low as 1024 MB of system memory would still waste resources even with the higher workloads like the unit test set. Considering that most of the environments will allocate 2048 megabytes or more, the inefficiency of the VM approach in terms of memory is obvious. Between Docker and Nix environments, there were no significant differences in memory usage.

7.3.3 Disk usage

On contrary to memory, disk can be used dynamically by virtual machines. In VirtualBox, a dynamically allocated disk image file is used for this purpose. The file will grow in size as the guest writes data to new sectors on its virtual hard disk or until the defined maximum size for the virtual storage is reached [25]. Vagrant boxes often use dynamically allocated disk images, but the maximum disk size might be as low as 10 GB like in the Debian jessie box. Many boxes also use VMDK i.e. Virtual Machine Disk -format, which cannot be resized with VirtualBox. It is possible however to first convert the VMDK to VDI or VirtualBox Disk Image, resize it and convert back to VMDK. When resizing, one has to be aware of the partitioning in the guest OS, as normally there is only one major partition fixed to the original disk size. In the example Debian development VM, the overall disk space used is about 3.1 GB. Most of the space is consumed by the 940 MB base VMDK image saved by Vagrant and the 2.2 GB VMDK image VirtualBox is actually using when running the VM. There are two reasons why the disk used by VirtualBox is bigger than the base. First is obviously all the software installed in the provisioning; VirtualBox Guest Additions, MongoDB and Node.js with all the project-related modules. The second reason is that when using a dynamically allocated disk, writing data to new sectors increases the size of the disk image. However, the size does never decrease, even when files are deleted from the disk. Hence, temporary files being written during the installation will make the disk image slightly larger. The disk image has

to be shrunk with a separate process of writing zeroes and compacting the disk in order to reduce the size.

Docker containers can dynamically use available disk from the host, with and without specifying a maximum size. All the container-data, including images and volumes are saved to respective directories on the host. Docker uses filesystem layers when operating on containers, and all the layers are being saved and not removed automatically. This means that when using Docker, disk space used is increasing all the time if system pruning is not executed [8]. The overall disk space used by the running container development environment is about 3.2 GB. As the Debian container image with all the software installed is 794 MB, MongoDB container image is 379 MB and the separate volume created for node modules is 533 MB, it is clear that a lot of the space is used for storing the layers created during the installation. Saving the layers is beneficial for caching purposes, but this design also demands disk space.

The overhead in disk usage with Nix comes from the fact that it uses its own packages instead of the system packages, unless one is using NixOS [23]. This means many of the core packages already installed on the operating system are duplicated as Nix components. However, Nix is very efficient in reusing the components in the store, so there is no overhead between components in different development environments. Compared to VMs and containers where software components are installed to respective isolated filesystems, Nix can reuse the same components in Nix store in multiple environments. The overall size of the example Nix development environment is about 1.7 GB. Nix core packages are 224 MB in size and Node.js and MongoDB packages 706 MB combined. Rest of the space is taken by node modules and MongoDB data files.

Table 7.1 shows the approximate resource usages of all the categories for the presented isolation solutions.

Solution	CPU usage (avg.)	Memory usage (avg.)	Disk usage
VM	18 %	1.2 GB (2 GB reserved)	3.1 GB
Containers	12 %	0.7 GB	3.2 GB
Nix	12 %	0.7 GB	1.7 GB

Table 7.1 *Resource usage of the isolation solutions*

7.4 Usability

Comparing usability of the approaches, usability is divided into three categories; learning curve, development workflow and maintenance load. The amount of effort needed in each of the phases with different isolation solutions is discussed shortly in this section.

7.4.1 Learning curve

With Vagrant, creating and provisioning new environments is simple and straightforward. Provisioning a virtual machine to a development environment is the same process one would do when setting up a native environment from the command-line, so there is no extra learning required on that part. A basic knowledge of virtualization and Vagrant are enough to start creating a development environment.

A containerized development environment brings some additional things to consider. The division of software components to containers makes the setup slightly more complex than using VMs. Instead of using a single Vagrantfile to provision a VM, multiple Dockerfiles, Docker compose -file and possibly multiple different container images are needed. Each container image might have a different set of tools and commands available, which adds to the variability of a containerized environment. Many times containerization forces a different kind of architecture and legacy applications might need some redesign to be able to run in containers. As isolation is done on a higher, more fine-grained level with containers, there is a higher amount of supporting functionality, e.g. network abstractions and shared volumes needed for the environment. Docker and Docker Compose can handle a lot of the supporting functionality between containers, but one will have to learn how to use them when designing the environment.

Nix is at a definitive disadvantage what it comes to the learning curve. Having its own, unique expression language and build system creates a steep learning curve. Knowledge on conventional package management systems and environment setups do not apply with Nix. Background on functional programming helps, but Nix still has its unique characteristics that one needs to be aware of. The initial amount of effort needed is much bigger when using Nix to create development environments, but when familiar with the syntax and structure, creating new environments is quite simple and effective.

7.4.2 Development workflow

Development in a dedicated VM with a full OS and GUI is identical to a native environment, apart from the slight performance degradation mentioned in section 5.1.1. Developing through a shared folder with a headless VM will require some extra configuration depending on the environment. For example, setting up remote debugging from an IDE requires port forwarding from the VM to the host. Also, using the shared folder can lead to problems in some setups, like discussed in section 7.1. In general, when the application is running in a VM, development workflow is simple and similar to working with a native environment. A VM is slower to start up than other solutions, but a 20-30 second delay once or a couple of times per day is not a big problem.

In a containerized environment, there are benefits from the different scope of isolation. It makes the environment customizable in smaller parts for quick testing of different versions and configurations. One can e.g. install different versions of software and add tools to one container being sure it does not affect the other containers. These kinds of targeted actions are not possible in a VM setup using global namespaces. This is a big advantage in development workflow in favor of the containerized environment. Separation of services to containers has its advantages, but it also brings some additional complexity to the workflow. Debugging code running in a container can be both easier due to separation of concerns and more complicated due to namespace separation and limited set of tools in the container. For example, checking logs and system state from containers might be harder because proper content viewing tools are missing. Also, file access between the host and containers is more complex because of the isolated filesystem. The remote debugging from an IDE can be configured similarly to the VM approach with the shared folder.

Nix does not virtualize anything so it creates a native development environment, but it has its own characteristics that one will have to comply to. Nix installs every component to the dedicated read-only path on a filesystem, which creates compatibility issues with some software, especially with other package managers that are trying to install packages to Nix store. When using Nix, other package managers should be avoided when possible, and if not, only local installations with those should be used.

7.4.3 Maintenance load

Maintaining the development environment is the third aspect to consider. Maintenance load in this context means the amount of effort needed to keep the envi-

ronments in sync across developers and the amount of effort needed when making changes to the environment. First off, like discussed in section 7.2, both VM and Docker environments suffer from the reproducibility issues. Unless one is always using exact version of packages and also controlling the repositories for those packages, the environments across developers will vary depending on the time of creation. When relying on third-party repositories, frequent recreation will keep the development environments in sync, but this can introduce additional maintenance work when the version one has pinned down does not exist anymore, or if using the latest available version, when the newly updated package has some differences in functionality. As discussed earlier, Nix environments are always reproducible, so the maintenance load in this respect is smaller.

Another aspect of maintenance, making changes to the environment, is simple in all of the approaches. Editing or adding lines in Vagrantfile or Dockerfile and recreating the environment. In Nix, the environment is not a mutable object that is recreated, it is just a shell environment started with different parameters. Nix-shell is reconfigured by editing the shell.nix -file. One thing to notice is that although ill-advised when pursuing parity, people tend to customize their development VMs, especially ones with a full operating system and a graphical user interface. If all the customizations are not part of the provisioning scripts like they should, recreating the environment after a common change might not sound appealing to all the developers. Small changes can obviously be done without recreation, but then the responsibility is on each developer and this might lead to disparity between environments.

7.5 Support and availability

Vagrant and Docker support all the major operating systems; Linux, macOS and Windows. They are both very popular technologies and have large ecosystems available. Being based on using common operating systems and their native package managers, the availability of software packages is not a problem with these approaches. Also, the amount of information and support in the community is extensive for Vagrant and Docker.

Nix is lacking in terms of support and availability compared to Vagrant and Docker. Nix officially supports Linux and macOS. Windows is not supported, but there are some working experiments on Windows also [38]. Nix is not a common package manager and its user base is very small compared to common ones. This inevitably shows in the number of available packages, information and support. Although Nix Packages collection has 42500 available packages (in July 2018), it is a marginal

number in relation to the amount of traditional Linux packages. Creating own packages with Nix is fairly simple when mastered, but unless Nix is going to be used also for building the software, creating own packages just to get a working development environment is not very productive. Also, when running into problems with the development environment setup, it might be hard to find support. Nix has an active community of contributors [13], but it is still a comparatively small community, so the effectiveness of the ecosystem is limited.

8. CONCLUSIONS

Virtualization combined with tools like Vagrant and Docker and their ecosystems makes creating and managing development environments simple. The isolation provided by virtualization is beneficial for development work, but sometimes a lighter solution providing weaker isolation with less overhead might also be an option. Comparing the solutions presented, the VM approach with Vagrant is arguably the easiest choice. The isolation provided by the VM is simple and effective, and because the environment is mostly set up and used similarly to a native environment, there is not much additional learning required. On the downside, a VM will have the largest resource overhead, especially in memory usage. Docker containers introduce a finer-grained virtualization, which makes the environment slightly more complex, but also more useful for development and testing. The modularity and separation of concerns achieved by containerization is an advantage that is hard to compete with. Docker containers use system resources dynamically without wasting capacity, but will use a decent amount of disk space because of the layered design. Reproducibility is an issue with Docker and other systems using traditional package managers. Inconsistent image tagging conventions and fluid repositories make true reproducibility hard to achieve.

Nix does not provide isolation like VMs and containers, it is a package manager with an architecture that enables creating environments with a specified set of isolated software components. It is useful for quickly setting up environments with non-conflicting components and tools. True reproducibility, non-conflicting dependencies and side-by-side installation of multiple versions of a package are some of the key advantages in using Nix for setting up development environments. Nix by itself does not provide any process, file system or network isolation so its environment separation from the host and other projects is limited. Often, for a development setup, the aforementioned namespace isolations are not necessary and component isolation provided by Nix is adequate. Nix also has a full ecosystem built around the package manager, which can be used to better utilize all the benefits associated with Nix. In addition to Nix and Nixpkgs, it includes the NixOS operating system, Hydra CI tool and NixOps deployment and provisioning tool. However, moving to the Nix ecosystem would be a major change and probably not a viable option for

most. [22]

There are many factors involved in choosing a development setup, e.g. host OS, application build and runtime requirements and developers' preferences. Also, development is only a part of a bigger process of delivering software. The design of the development environment should be in line with the whole delivery process. Ideally development should be done in an environment simulating production as closely as possible. Nowadays software is being run more and more on containers, which are run on top of either virtual machines or directly on physical machines i.e. bare metal. This advocates using containers in development, as production-like setup should be the goal. There are situations where containers are not a viable option by themselves, and virtual machines are needed. One example is a project where the target application and necessary tools need a different operating system than the host machine, e.g. Linux environment is needed on a Windows host. Another would be a project which requires a set of graphical tools, as VMs are usually more fitting to graphical applications.

None of the isolation solutions presented in this work rule each other out. One can use Docker containers inside VMs and Nix inside VMs and containers. A typical example of using containers inside a VM for development purposes is a Linux development VM on a Windows host, which is using containerized environments in development. One might also use a nested combination of a VM and containers even if the host and guest OSs are the same. Installing Nix to a VM or to a container makes sense when reproducibility is important, but at the same time there is a need for a deeper level of isolation. With Docker containers, one can create a useful setup by sharing the Nix store on the host machine with multiple containers through a shared volume. This way the same Nix components can be used in all the containers.

Overall, finding the right development setup and isolation method for a project is usually a process which requires trying out different approaches. It is hard to take all the variables into account beforehand and there might be factors that rule out some of the options. Perfect solutions are rarely found, so a balance between pros and cons needs to be figured out. Luckily, there are lots of tools and technologies available, so there should be a working solution for every project.

BIBLIOGRAPHY

[1] CentOS, "vault.centos.org." [Online]. Accessed: 4.8.2018 Available: http://vault.centos.org

[2] J. Cormack, "Announcing linuxkit: A toolkit for building secure, lean and portable linux subsystems." [Online]. Accessed: 4.8.2018 Available: https://blog.docker.com/2017/04/introducing-linuxkit-container-os-toolkit

[3] P. J. Denning, "Virtual memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, Sep. 1970. [Online]. Accessed: 4.8.2018 Available: http://doi.acm.org/10.1145/356571.356573

[4] Docker Inc., "Docker Docs." [Online]. Accessed: 4.8.2018 Available: https://docs.docker.com/docker-for-mac/

[5] Docker Inc., "Docker Docs." [Online]. Accessed: 4.8.2018 Available: https://docs.docker.com/engine/docker-overview/#the-underlying-technology

[6] Docker Inc., "Docker Docs." [Online]. Accessed: 4.8.2018 Available: https://docs.docker.com/compose/overview

[7] Docker Inc., "Docker Docs." [Online]. Accessed: 4.8.2018 Available: https://docs.docker.com/engine/reference/commandline/pull/#pull-an-image-by-digest-immutable-identifier

[8] Docker Inc., "Docker Docs." [Online]. Accessed: 4.8.2018 Available: https://docs.docker.com/engine/reference/commandline/system_prune

[9] E. Dolstra, "The purely functional software deployment model," 2006. [Online]. Accessed: 4.8.2018 Available: https://nixos.org/~eelco/pubs/phd-thesis.pdf

[10] E. Dolstra *et al.*, "Nixos: A purely functional linux distribution," *SIGPLAN Not.*, vol. 43, no. 9, pp. 367–378, Sep. 2008. [Online]. Accessed: 4.8.2018 Available: http://doi.acm.org/10.1145/1411203.1411255

[11] C. Ebert *et al.*, "Devops," *IEEE Software*, vol. 33, no. 3, pp. 94–100, May 2016.

[12] W. Felter *et al.*, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172.

[13] GitHub Inc., "GitHub Repository Insights." [Online]. Accessed: 4.8.2018 Available: https://github.com/NixOS/nixpkgs/pulse

[14] HashiCorp, "Vagrant docs - box versioning." [Online]. Accessed: 4.8.2018 Available: https://www.vagrantup.com/docs/boxes/versioning.html

[15] HashiCorp, "Vagrant docs - creating a base box." [Online]. Accessed: 4.8.2018 Available: https://www.vagrantup.com/docs/boxes/base.html

[16] HashiCorp, "Vagrant docs - synced folders." [Online]. Accessed: 4.8.2018 Available: https://www.vagrantup.com/docs/synced-folders/basic_usage.html#symbolic-links

[17] S. Johnston, "Docker announces commercial partnership with microsoft to double container market by extending docker engine to windows server." [Online]. Accessed: 4.8.2018 Available: https://blog.docker.com/2016/09/docker-microsoft-partnership

[18] Linux VServer, "Linux VServer whitepaper." [Online]. Accessed: 4.8.2018 Available: http://linux-vserver.org/Paper

[19] Microsoft Corporation, "Hyper-v technology overview." [Online]. Accessed: 4.8.2018 Available: https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview

[20] mist64, "xhyve." [Online]. Accessed: 4.8.2018 Available: https://github.com/mist64/xhyve

[21] Moby, "Hyperkit." [Online]. Accessed: 4.8.2018 Available: https://github.com/moby/hyperkit

[22] NixOS, "Nix Ecosystem." [Online]. Accessed: 4.8.2018 Available: https://nixos.wiki/wiki/Nix_Ecosystem

[23] NixOS, "Nix Package Manager Guide." [Online]. Accessed: 4.8.2018 Available: https://nixos.org/nix/manual

[24] opensourcesurvey.org, "Open Source Survey." [Online]. Accessed: 4.8.2018 Available: http://opensourcesurvey.org/2017/#insights

[25] Oracle Corporation, "Oracle vm virtualbox user manual." [Online]. Accessed: 4.8.2018 Available: http://download.virtualbox.org/virtualbox/5.2.4/UserManual.pdf

[26] Oracle Corporation, "Oracle vm virtualbox user manual - guest additions." [Online]. Accessed: 4.8.2018 Available: https://www.virtualbox.org/manual/ch04.html

[27] G. J. Popek *et al.*, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974. [Online]. Accessed: 4.8.2018 Available: http://doi.acm.org/10.1145/361011.361073

[28] Portworx Inc., "Portworx annual container adoption survey 2017," 2017. [Online]. Accessed: 4.8.2018 Available: https://portworx.com/wp-content/uploads/2017/04/Portworx_Annual_Container_Adoption_Survey_2017_Report.pdf

[29] Red Hat, Inc., "Introduction to linux containers." [Online]. Accessed: 4.8.2018 Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers

[30] collinsdictionary.com, "Collins english dictionary." [Online]. Accessed: 4.8.2018 Available: https://www.collinsdictionary.com/dictionary/english/virtualize

[31] J. S. Robin *et al.*, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," 2000. [Online]. Accessed: 4.8.2018 Available: https://calhoun.nps.edu/handle/10945/7100

[32] F. Rodrguez-Haro *et al.*, "A summary of virtualization techniques," *Procedia Technology*, vol. 3, pp. 267 – 272, 2012, the 2012 Iberoamerican Conference on Electronics Engineering and Computer Science. [Online]. Accessed: 4.8.2018 Available: http://www.sciencedirect.com/science/article/pii/S2212017312002587

[33] semver.org, "Semantic versioning 2.0.0." [Online]. Accessed: 4.8.2018 Available: https://semver.org/

[34] svanderburg, "node2nix." [Online]. Accessed: 4.8.2018 Available: https://github.com/svanderburg/node2nix

[35] Systutorials, "Linux Man Pages." [Online]. Accessed: 4.8.2018 Available: http://man7.org/linux/man-pages/man7/namespaces.7.html

[36] The FreeBSD Project, "FreeBSD Docs." [Online]. Accessed: 4.8.2018 Available: https://www.freebsd.org/doc/handbook/jails.html

[37] The FreeBSD Project, "FreeBSD Manual Pages." [Online]. Accessed: 4.8.2018 Available: https://www.freebsd.org/cgi/man.cgi?query=chroot&sektion=2&n=1

[38] tweag.io, "Nix on the windows subsystem for linux." [Online]. Accessed: 4.8.2018 Available: https://www.tweag.io/posts/2017-11-10-nix-on-wsl.html

[39] VMWare Inc., "Understanding Full Virtualization, Paravirtualization, and Hardware Assist," 2008. [Online]. Accessed: 4.8.2018 Available: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf

[40] P. Vohra *et al.*, "A Contrast and Comparison of Modern Software Process Models," *International Journal of Computer Applications*, 2013. [Online]. Accessed: 4.8.2018 Available: https://research.ijcaonline.org/icamt/number1/icamt1013.pdf

[41] Wikipedia , "List of software package management systems." [Online]. Accessed: 4.8.2018 Available: https://en.wikipedia.org/wiki/List_of_software_package_management_systems

[42] Xebia, "Deep dive into windows server containers and docker part 2 underlying implementation of windows server containers." [Online]. Accessed: 4.8.2018 Available: http://blog.xebia.com/deep-dive-into-windows-server-containers-and-docker-part-2-underlying-implementation-of-windows-server-containers

[43] Xebia, "Deep dive into windows server containers and docker part 3 underlying implementation of hyper-v containers." [Online]. Accessed: 4.8.2018 Available: http://www.solidalm.com/2018/03/10/hyper-v-and-linux-containers-on-windows